

A Structured Overlay for Non-uniform Node Identifier Distribution Based on Flexible Routing Tables

Takehiro Miyao, Hiroya Nagao, Kazuyuki Shudo
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, JAPAN
Email: {takehiro.miyao, hiroya.nagao, shudo}@is.titech.ac.jp

Abstract—A large fraction of structured overlays work efficiently as long as node identifiers follow a uniform distribution with high probability. There is another kind of structured overlay supporting non-uniform node identifier distributions and it enables a DHT to support range queries. This paper presents FRT-Chord[#], such a structured overlay for non-uniform node identifier distributions. It is based on Flexible Routing Tables (FRT), a method for designing structured overlays, and inherits advantageous features of FRT, that existing overlays do not hold. Such features include extensibility, arbitrary routing table capacity.

I. INTRODUCTION

There are a number of structured overlays, routing algorithms for peer-to-peer, proposed. A large fraction of structured overlays including initially proposed ones [1]–[5] were designed to achieve efficient lookups in case node identifiers follow a uniform distribution with high probability. They do not work efficiently otherwise.

However, there are applications that require structured overlays for non-uniform node identifier distributions. Range queries over a Distributed Hash Table (DHT) is one of such applications. Range queries require assigning continuous identifiers to data with continuous keys to achieve efficient lookups. Otherwise continuous keys are distributed to numerous nodes, that a range query has to involve. Such a continuous assignment leads to a non-uniform data identifier distribution.

In case data identifiers are distributed non-uniformly, load balance is a problem. Virtual nodes is one of solutions to the problem. We focus on another promising solution, that is to make a node identifier distribution follow a data identifier distribution. To achieve it, node identifiers can be adjusted by leave and rejoin of nodes. Here a structured overlay supporting non-uniform node identifier distributions achieves range queries and load balance.

There are structured overlays for non-uniform node identifier distributions proposed [7], [8]. Such overlays enable a DHT supporting range queries, and support efficient lookups with a small number of nodes. With few nodes, consistent hashing [6] and other hashing techniques for determining node

identifiers leads to non-uniform node identifier distributions. Furthermore, such overlays enable arbitrary assignments of node identifiers. For example, node identifiers can reflect network proximity.

This paper presents FRT-Chord[#], a structured overlay for non-uniform node identifier distributions. FRT-Chord[#] is based on Flexible Routing Tables (FRT) [9], a method for designing structured overlays, and thus inherits advantageous features of FRT, that the existing overlays do not hold. Such features include extensibility [9], [10], arbitrary routing table capacity, and one-hop property [11]. GFRT [9] and PFRT [10] are examples of extensions to FRT-based structured overlays. GFRT is an extension to consider node groups and PFRT is for network proximity. FRT-Chord[#] itself does not perform exact 1 hop routing, but it achieves the lowest route length, that is 2, in case the number of node is less than routing table capacity thanks to the one-hop property of FRT.

Experimental results show that FRT-Chord[#] works efficiently with non-uniform node identifier distributions as well as with a uniform distribution.

This paper is structured as follows. Section II presents prior knowledge by introducing related work. Section III describes FRT-Chord[#]. Section IV demonstrates the properties of FRT-Chord[#] including adaptability to non-uniform node identifier distributions and arbitrary routing table capacity inherited from FRT. In Section V, we summarize our contributions.

II. RELATED WORK

A. Chord

Chord [1] is one of initially proposed structured overlays. Chord was designed to achieve efficient lookups under conditions where node identifiers follow a uniform distribution with high probability.

Chord assigns an m -bit identifier to a node by hashing the node's address. The identifier space in Chord has a ring structure, with identifier distance $d(x, y)$ between identifiers x and y in a clockwise direction in the ring calculated as

$$d(x, y) = \begin{cases} y - x, & x < y, \\ y - x + 2^m, & y \leq x. \end{cases} \quad (1)$$

This work was supported by JSPS KAKENHI Grant Numbers 25700008 and 24650025.

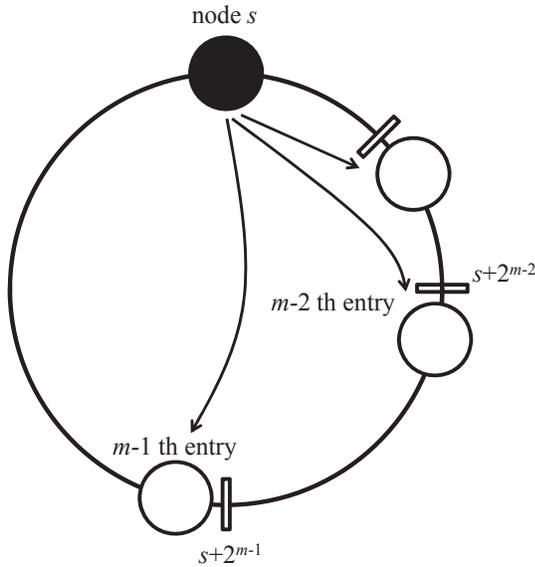


Fig. 1. Finger table

The first node that an identifier reaches while progressing clockwise is called the successor node.

Each entry in a routing table holds a node's identifier and its address, but in this paper an entry is sometimes expressed as the node pointed to by the entry. Routing tables in Chord consist of three parts: a successor list, a predecessor, and a finger table. The successor list of a node contains a certain number u closest nodes from the node in the identifier space. The successor list is used for reaching destination nodes. The predecessor is the farthest node from the node in the identifier space. The finger table is used to decrease route lengths. The i th entry ($i = 0, 1, 2, \dots, m-1$) of the finger table for a node s contains the successor node of the identifier $s.id + 2^i$ (Fig. 1). Thus, if N is the number of nodes, then the number of entries in a finger table is about $\log N$.

In Chord, *stabilize* and *notify* functions guarantee reachability. All nodes periodically execute *stabilize* to search for their current successors. When a node finds a new successor, it executes *notify* to instruct the successor to update its predecessor. A node periodically executes *fix_fingers* function to ensure its finger table entries are correct. Lookups are performed for identifier $s.id + 2^i$ ($i = 0, 1, 2, \dots, m-1$) in *fix_fingers*, and the node updates the i th entry of the finger table.

When routing, a request message arrives at the destination node by greedy routing, which selects a forwarding node from the routing table as follows. The message is repeatedly forwarded from each node to the neighbor closest to the destination identifier in the node's routing table. Route lengths in Chord are $O(\log N)$ -hops when node identifiers follow a uniform distribution with high probability.

B. FRT-Chord

Flexible Routing Tables (FRT) [9] is a method for designing structured overlays. FRT-based algorithms have various advantageous features, such as extensibility, arbitrary routing table

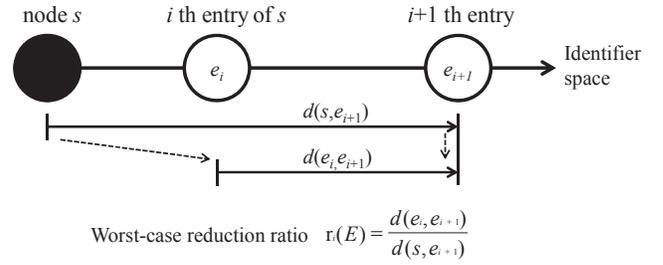


Fig. 2. Worst-case reduction ratio

capacity, and one-hop property.

FRT-Chord [9] uses the same identifier space as Chord does. However, a node s in FRT-Chord maintains only a single routing table E containing entries e_i ($i = 1, 2, \dots, |E|$), which are sorted in ascending order of identifier distance from s ($i < j \Rightarrow d(s, e_i) < d(s, e_j)$). Thus, $\{e_i\}_{i=1,2,\dots,u}$ is the successor list of s , where e_1 is s 's successor, u is the capacity of the successor list, and $e_{|E|}$ is s 's predecessor. Since routing tables in FRT-based algorithms are maintained by using a total order \leq_{RT} on the set of all routing table patterns, designers need to define the order.

1) *Total Order on Routing Table Set*: Routing tables are updated according to \leq_{RT} , which is a total order on the set of all routing table patterns. $E \leq_{RT} F$ then expresses that routing table E is better than F , and this determination is based on identifier distance between nodes. The order in FRT-Chord is defined as follows.

A node s calculates the worst-case reduction ratio $r_i(E)$ (Fig. 2) of a forwarding to a node e_i other than its predecessor. The reduction ratio is the fraction by which the remaining identifier distance is reduced when s forwards a message to each entry e_i :

$$r_i(E) = \frac{d(e_i, e_{i+1})}{d(n, e_{i+1})} \quad (i = 1, 2, \dots, |E| - 1). \quad (2)$$

Let $\{r_{(i)}(E)\}$ be the list in which e_i are ranked in descending order by $\{r_i(E)\}$. Then,

$$E \leq_{RT} F \iff \{r_{(i)}(E)\} \leq_{dic} \{r_{(i)}(F)\}, \quad (3)$$

where the lexicographical order \leq_{dic} is defined as follows:

$$\{a_i\} <_{dic} \{b_i\} \iff a_k < b_k \quad (k = \min\{i | a_i \neq b_i\}), \quad (4)$$

$$\{a_i\} =_{dic} \{b_i\} \iff a_i = b_i, \quad (5)$$

$$\{a_i\} \leq_{dic} \{b_i\} \iff (a_i <_{dic} b_i) \cup (a_i =_{dic} b_i). \quad (6)$$

2) *Guarantee of Reachability*: These operations guarantee reachability in FRT. *Guarantee of reachability* function in FRT-Chord relies on the same as *stabilize* and *notify* in Chord.

3) *Entry Learning*: This function obtains a node's information and inserts it into a routing table in FRT. A node obtains another node's information in the following situations.

- When a new node joins the network in FRT-Chord, it obtains information on its successor and the information in the successor's routing table.

- When node s communicates with node a , s obtains a 's information.
- Each node periodically executes *active learning lookups* function to obtain information on other nodes.

Active learning lookups in FRT-Chord actively acquires node information, and is similar to *fix_fingers* in Chord. When a node s executes *active learning lookups*, a lookup is performed for the identifier k calculated by

$$k = s.\text{id} + d(s, e_1) \left(\frac{d(s, e_{|E|})}{d(s, e_1)} \right)^{\text{rnd}}, \quad (7)$$

where rnd is a random number between 0 and 1. s then obtains the node information on k 's successor.

4) *Entry Filtering*: If the number of entries $|E|$ in the routing table E is greater than the capacity L of the routing table, then some entries should be removed from E . *Entry filtering* is the function that removes an entry in FRT. The function determines the entry for removal e_r according to \leq_{RT} , as follows:

$$E \setminus \{e_r\} \leq_{\text{RT}} E \setminus \{e\}, \quad \forall e \in E. \quad (8)$$

By calculating a canonical spacing S_i^E for each entry e_i in E , e_r can be found efficiently:

$$S_i^E = \log \frac{d(s, e_{i+1})}{d(s, e_i)}. \quad (9)$$

If $S_{i'}^E + S_{i'}^E$ is the minimum value of $S_{i-1}^E + S_i^E$ for e_i , then $e_r = e_{i'}$. In FRT, any entry that must not be removed from the routing tables is called a sticky entry. Sticky entries in FRT-Chord consist of the successor list and predecessor entries that are required to reach a destination node.

The steps in *entry filtering* are as follows. Let C be the set of candidates for e_r .

- 1) Add all entries in E to C .
- 2) Remove the sticky entries from C .
- 3) When $S_{i'}^E + S_{i'}^E$ is found for $e_i \in C$, e_r equals $e_{i'}$ and is removed from E .

By sorting $\{S_{i-1}^E + S_i^E\}$ into ascending order, e_r can be found in $O(1)$ steps.

C. Chord[#]

Chord[#] [7] is a structured overlay. Chord[#] was designed to achieve efficient lookups even when node identifiers do not follow a uniform distribution.

Chord[#] uses the same identifier space as Chord and uses greedy routing. Routing tables in Chord[#] are similar to those in Chord, but the finger tables are different. Chord builds finger tables on the basis of identifier distance between any two nodes. Therefore, the route lengths in Chord are larger when node density in an identifier range is high. However, Chord[#] builds finger tables on the basis of the number of nodes between any two nodes. Thus, the route lengths in Chord[#] are not long, even when node density is high.

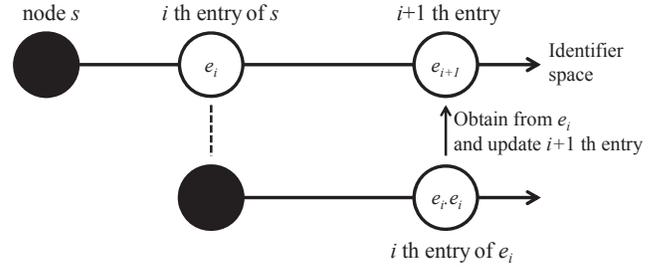


Fig. 3. Finger table in Chord[#]

An entry e_i has not only the entry's identifier $e_i.\text{id}$ and address $e_i.\text{addr}$, but also the i th entry e_{i, e_i} of e_i 's finger table. A finger table consists of entries as follows (Fig.3):

$$e_i = \begin{cases} \text{successor}, & i = 0, \\ e_{i-1, e_{i-1}}, & i > 0. \end{cases} \quad (10)$$

In Chord[#], each node periodically executes *stabilize* and *notify* to guarantee reachability, as in Chord. Each node periodically executes *fix_fingers* to make sure its finger table entries are correct. *Fix_fingers* in Chord[#] is different from that in Chord, in that the node obtains the i th entry of e_i 's finger table from e_i and updates the $i + 1$ th entry of its own finger table.

III. FRT-CHORD[#]

We presented FRT-Chord[#], which is a structured overlay for non-uniform node identifier distribution. FRT-Chord[#] is an FRT-based algorithm, and so inherits advantageous features of FRT.

FRT-Chord[#] uses the same identifier space as Chord and builds routing tables with neighbor nodes' routing tables, so an entry e has the entry's identifier $e.\text{id}$, address $e.\text{addr}$ and routing table $e.\text{RT}$.

A. Total Order on Routing Table Set

To design FRT-Chord[#], we define a total order \leq_{RT} on a routing table set. First, let $n_s(x)$ be the number of entries from $s.\text{id}$ to an identifier x . FRT-Chord[#] uses an identifier space with a ring structure. Therefore,

$$n_s(x) = \max\{i | d(s, e_i) \leq d(s, E_s.\text{forward}(x)), e_i \in E_s\} \quad (11)$$

holds, where $E_s.\text{forward}(x)$ is the entry in routing table E_s to which node s forwards the message of destination identifier x . When node s forwards to an entry e_i ,

$$E_s.\text{forward}(x) = e_i \quad (12)$$

holds, so

$$n_s(x) = i \quad (13)$$

also holds. When $n_s(x) = 0$ holds, node s is the determination node of identifier x .

Next, let $f_s(x)$ be the reduction value in the number of entries in $[s, x)$ by a forwarding.

$$f_s(x) = n_s(x) - n_{E_s.\text{forward}(x)}(x) \quad (14)$$

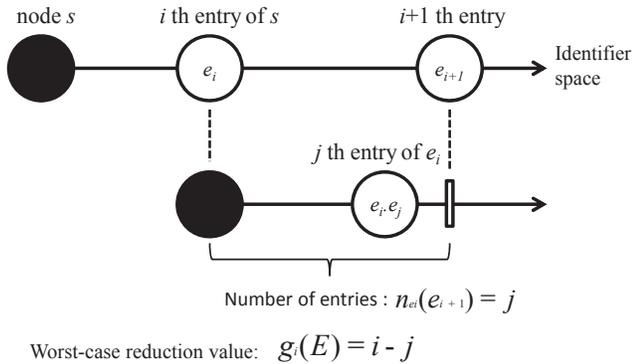


Fig. 4. Worst-case reduction value

holds because (12) and (13) hold. Every forwarding reduces $n_s(x)$ of each self node s so that $n_s(x)$ becomes 0. Therefore, the higher $f_s(x)$ is, the better entry e_i is.

Finally, let $g_i(E)$ be the worst-case reduction value in the number of entries when forwarding to entry e_i (Fig. 3). A node evaluates an entry e_i by $g_i(E)$. $g_i(E)$ takes the worst-case value when $x = e_{i+1}.id$ holds, so

$$g_i(E) = f_s(e_{i+1}) \quad (15)$$

holds. Because (14) holds,

$$g_i(E) = n_s(e_{i+1}) - n_{E_s.forward(e_{i+1})}(e_{i+1}) \quad (16)$$

also holds. Since FRT-Chord[#] uses a cyclic identifier space and greedy routing,

$$E_s.forward(e_{i+1}) = e_i \quad (17)$$

holds. Therefore,

$$g_i(E) = n_s(e_{i+1}) - n_{e_i}(e_{i+1}) \quad (18)$$

also holds. Because (13) holds,

$$g_i(E) = i - n_{e_i}(e_{i+1}) \quad (19)$$

follows. The entry e_i has the routing table of e_i so node s can compute $g_i(E)$ for e_i .

$E \leq_{RT} F$ expresses that routing table E is better than F . When $g_i(E)$ is higher, the entry e_i gets a higher score. Therefore, we define the total order on a routing table set as follows:

$$E \leq_{RT} F \Leftrightarrow \{g_{(i)}(F)\} \leq_{dic} \{g_{(i)}(E)\}. \quad (20)$$

Then, $\{g_{(i)}(E)\}$ is the list in which e_i are ranked in descending order by $\{f_i(E)\}$.

B. Guarantee of Reachability

Guarantee of reachability in FRT-Chord[#] is the same as that in FRT-Chord.

C. Entry Learning

Entry learning in FRT-Chord[#] is the same as that in FRT-Chord.

D. Entry Filtering

If the number of entries $|E|$ in the routing table E is greater than the capacity L of the routing table E , then some entries should be removed from E . *Entry filtering* removes an entry according to \leq_{RT} . Sticky entries in FRT-Chord[#] are the same as in FRT-Chord.

The steps in *entry filtering* are as follows. Let C be the set of candidates for e_r .

- 1) Add all entries in E to C .
- 2) Remove the sticky entries from C .
- 3) When $E \setminus \{e_r\} \leq_{RT} E \setminus \{e\}$ holds for each $e \in C$, e_r is removed from E .

By sorting $\{E \setminus \{e_i\}\}_{i=1,2,\dots}$ into the total order of the routing table set, e_r can be found in $O(1)$ steps. The number of $\{E \setminus \{e_i\}\}$ is L so sorting $\{E \setminus \{e_i\}\}$ has computational complexity $O(L \log L)$. Comparison of two routing tables is in $O(L)$ because of using lexicographical ordering. Therefore, e_r can be found in $O(L^2 \log L)$ steps.

IV. EVALUATION

In this section, we evaluate FRT-Chord[#] through simulation. We implemented a DHT using FRT-Chord[#] in Overlay Weaver [12], [13], which is an overlay construction toolkit, and performed experiments on the following machine.

- Simulator: Overlay Weaver 0.10.1
- Operating system: Linux 2.6.35.10-74.fc14.x86_64
- Central processing unit: Intel Xeon E5620 (2.40 GHz)
- Java virtual machine: Java SE 6 Update 22

Nodes are assigned identifiers so that node identifiers follow either a Zipf distribution ($\alpha = 0.95, 0.7$) or a uniform distribution.

A. Comparison with other algorithms

We measured route length in Chord, FRT-Chord and FRT-Chord[#] in simulations configured as follows:

- Number of nodes: 10000
- Successor list capacity: 4
- Capacity of routing tables: 16.

Because the number of nodes is 10000, the number of entries in a finger table in Chord is about 13; that is, the number of entries in all three parts of the routing tables in Chord is about 18. Therefore, the capacity of a routing table in FRT-Chord[#] is set at 16 so that FRT-Chord[#] does not have an advantage in the number of entries that can be held.

Fig. 5 shows the average and the 99th percentiles of route length in Chord, FRT-Chord, and FRT-Chord[#] when node identifiers follow a Zipf distribution ($\alpha = 0.7$). Node density of an identifier range in a Zipf distribution ($\alpha = 0.7$) is higher than in a uniform distribution. The average route length in FRT-Chord[#] is 7.01, the average in Chord is 7.67 and the average in FRT-Chord is 7.33. The 99th percentile of route length in FRT-Chord[#] is 12, the 99th percentile in Chord is 15, and the 99th percentile in FRT-Chord is 15.

Fig. 6 shows the average and 99th percentiles of route length in Chord, FRT-Chord, and FRT-Chord[#] when node identifiers

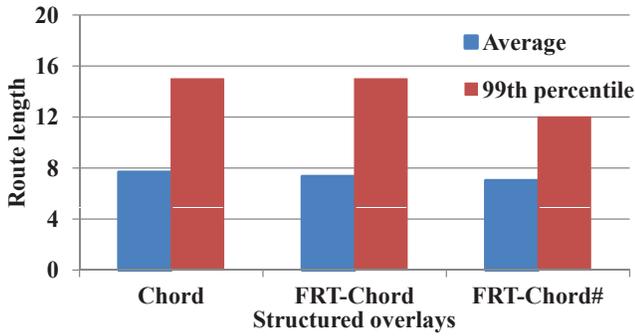


Fig. 5. Route lengths for Zipf distribution ($\alpha = 0.7$)

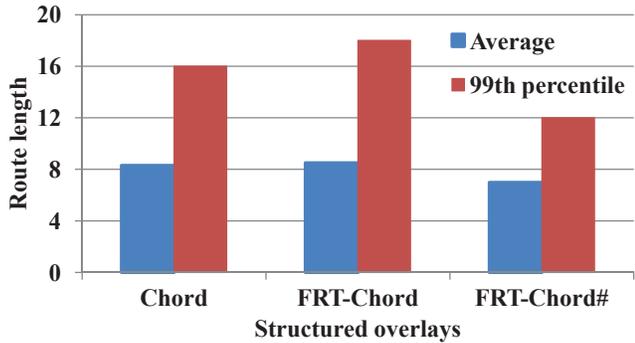


Fig. 6. Route lengths for Zipf distribution ($\alpha = 0.95$)

follow a Zipf distribution ($\alpha = 0.95$). Node density of an identifier range in a Zipf distribution with $\alpha = 0.95$ is higher than in a Zipf distribution with $\alpha = 0.7$. The average route length in FRT-Chord# is 6.98, the average in Chord is 8.30, and the average in FRT-Chord is 8.50. The 99th percentile of route length in FRT-Chord# is 12, the 99th percentile in Chord is 16, and the 99th percentile in FRT-Chord is 18. Route length in FRT-Chord# is less than in Chord and FRT-Chord when node identifiers follow a Zipf distribution ($\alpha = 0.95$). Figs. 5 and 6 show that route lengths in FRT-Chord# are less than in Chord and FRT-Chord when node identifier do not follow a uniform distribution.

Fig. 7 shows the average and 99th percentiles of route length in Chord, FRT-Chord and FRT-Chord# when node identifiers follow a uniform distribution. The average route length in FRT-Chord# is 6.97, the average in Chord is 7.21, and the average in FRT-Chord is 6.76. The 99th percentile of route length in FRT-Chord# is 12, the 99th percentile in Chord is 12, and the 99th percentile in FRT-Chord is 11. Route length in FRT-Chord# is only as large as in Chord and FRT-Chord, even if node identifiers follow a uniform distribution.

Figs. 5, 6 and 7 show that route lengths of FRT-Chord# in a uniform distribution are as large as in a Zipf distribution ($\alpha = 0.7, 0.95$). Therefore, FRT-Chord# is a structured overlay suitable for non-uniform distributions of node identifiers.

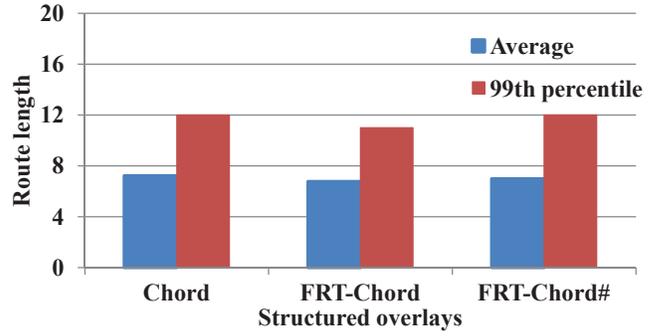


Fig. 7. Route lengths for uniform distribution

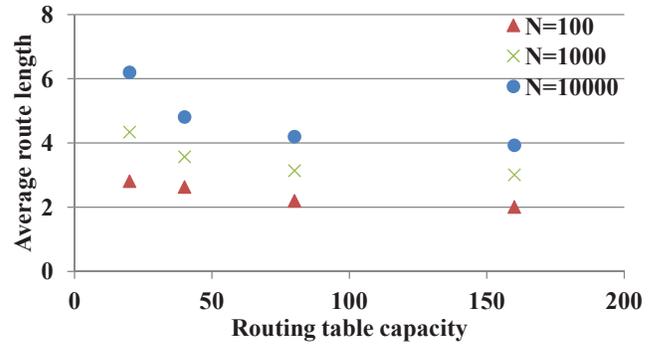


Fig. 8. Route lengths in each routing table size

B. Advantageous features of FRT

We examined whether FRT-Chord# retains one of beneficial features of FRT, such as arbitrary capacity of routing tables and one-hop property. We measured the average route length in FRT-Chord# under the following parameter values:

- Number of nodes: 100, 1000, and 10000.
- Capacity of each routing table: 20–160.

Fig. 8 shows the simulation results for the average route length. As the capacity of the routing tables increases, the average route length decreases. Therefore, FRT-Chord# can set arbitrary routing table capacities.

When the number of nodes N is less than the capacity L of each routing table, for example, when $N = 100$ and $L = 160$, the average route length is 2, which is the lowest route length of FRT-Chord# in usual cases. FRT-Chord# inherits the lowest route length as 2 from Chord. In Chord a route requires 2 hops at least to reach the responsible node. A route once reaches the predecessor of the target identifier and then steps to the responsible node. FRT-Chord# itself does not perform 1 hop routing, but it achieves the lowest route length thanks to the one-hop property of FRT.

V. CONCLUSION

This paper presented FRT-Chord#, an FRT-based structured overlay for non-uniform node identifier distributions. Structured overlays supporting non-uniform node identifier distributions enable a DHT supporting range queries, support

efficient lookups with a small number of nodes, and enable arbitrary assignment of node identifiers, for example, identifier assignment based on network proximity.

FRT-Chord[#] is based on Flexible Routing Tables (FRT), a method for designing structured overlays, and thus inherits advantageous features of FRT, that existing overlays do not hold. Such features include extensibility, arbitrary routing table capacity, and one-hop property.

Experimental results showed routing efficiency of FRT-Chord[#] with uniform and non-uniform node identifier distributions. In the non-uniform cases, FRT-Chord[#] showed better route length than structured overlays for uniform distributions. Even in the uniform case, the results are comparable to the other structured overlays. Another result supports that FRT-Chord[#] inherits one of features of FRT, arbitrary routing table capacity.

Future work includes comparison with Chord[#]. In a stable network, without churn, FRT-Chord[#] and Chord[#] will show similar route length. The difference between them is flexibility of FRT-Chord[#], in which a node can hold arbitrary nodes in its routing table. We expect that flexibility of FRT-based overlays results in adaptability to churn.

REFERENCES

- [1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.
- [2] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information systems based on the XOR metric," in *Proc. IPTPS '02*, MA, USA, 2002, pp. 53–65.
- [3] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, 2001, pp. 329–350.
- [4] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry," *IEEE J. Sel. Areas Commun.*, no. 1, pp. 41–53.
- [5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *ACM SIGCOMM Comput. Comm. Rev.*, vol. 31, no. 4, pp. 161–172, 2001.
- [6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory of Computing*. ACM, 1997, pp. 654–663.
- [7] T. Schütt, F. Schintke, and A. Reinefeld, "Range queries on structured overlay networks," *Comput. Commun.*, vol. 31, no. 2, pp. 280–291, 2008.
- [8] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4. ACM, 2004, pp. 353–366.
- [9] H. Nagao and K. Shudo, "Flexible Routing Tables: Designing routing algorithms for overlays based on a total order on a routing table set," in *Proc. IEEE P2P '11*, Kyoto, Japan, 2011, pp. 72–81.
- [10] T. Miyao, H. Nagao, and K. Shudo, "A method for designing proximity-aware routing algorithms for structured overlays," in *Proc. IEEE ISCC '13*, Split, Croatia, 2013.
- [11] Y. Ando, H. Nagao, T. Miyao, and K. Shudo, "FRT-2-Chord: A DHT supporting seamless transition between one-hop and multi-hop lookups with symmetric routing table," in *Proc. ICOIN 2014*, Phuket, Thailand, 2014.
- [12] K. Shudo, "Overlay Weaver." <http://overlayweaver.sourceforge.net/>.
- [13] K. Shudo, Y. Tanaka, and S. Sekiguchi, "Overlay Weaver: An overlay construction toolkit," *Comput. Comm.*, vol. 31, no. 2, pp. 402–412, 2008.